



Repositorio Institucional de la Universidad Autónoma de Madrid

<https://repositorio.uam.es>

Esta es la **versión de autor** de la comunicación de congreso publicada en:
This is an **author produced version** of a paper published in:

Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach:
First International Work-Conference on the Interplay Between Natural and Artificial
Computation, IWINAC 2005, Las Palmas, Spain, June 15-18, 2005, Proceedings.
Part II. Lecture Notes in Computer Science, Volumen 3562. Springer, 2005. 182-191.

DOI: http://dx.doi.org/10.1007/11499305_19

Copyright: © 2005 Springer-Verlag

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription

Attribute Grammar Evolution

Marina de la Cruz Echeanda, Alfonso Ortega de la Puente, Manuel Alfonseca
{marina.cruz, alfonso.ortega, manuel.alfonseca@uam.es}

Universidad Autnoma de Madrid. Departamento de Ingeniera Informtica

Abstract. This paper describes Attribute Grammar Evolution (AGE), a new Automatic Evolutionary Programming algorithm that extends standard Grammar Evolution (GE) by replacing context-free grammars by attribute grammars. GE only takes into account syntactic restrictions to generate valid individuals. AGE adds semantics to ensure that both semantically and syntactically valid individuals are generated. Attribute grammars make it possible to semantically describe the solution. The paper shows empirically that AGE is as good as GE for a classical problem, and proves that including semantics in the grammar can improve GE performance. An important conclusion is that adding too much semantics can make the search difficult.

1 Introduction

1.1 Syntax and semantics of high level programming languages

The differences between syntax and semantics in high level programming languages are rather artificial.

Turing Machines are associated to Chomsky 0 grammars [1], while the syntax of high level programming languages is usually expressed by means of context free grammars. Context free grammars are associated to pushdown automata, which have less expressive power than Turing Machines. The expressive power gap between Chomsky 0 and context free grammars is usually called the “semantics” of high level programming languages. This gap mainly deals with context dependent constructions, such as the mandatory declaration of the variables before their use or, the constraints about number and type of the arguments in functions calls, which must agree with their declaration.

1.2 Attribute grammars

Attribute grammars [2] are one of the tools used to describe high level programming languages completely (their syntax and their semantics). Attribute grammars extend context free grammars by adding these components to them:

- Each non terminal symbol has a set of attributes. Attributes are similar to the variables in programming languages; they have a name and their values belong to a given domain.
- Each rule contains expressions to compute the value of the attributes.

A detailed description of attribute grammars and some examples of their use can be found in references [3,4]

1.3 Grammar evolution (GE)

GE [5] is an automatic programming evolutionary algorithm independent of the target programming language, which includes a standard representation of genotypes as strings of integers (codons), and a context free grammar, as inputs for the deterministic mapping of a genotype into a phenotype. This mapping minimizes the generation of syntactically invalid phenotypes. Genetic operators act at the genotype level, while the fitness function is evaluated on the phenotypes.

The *genotype to phenotype mapping* is an algorithm that iterates on the string of codons and derives words by applying the context free grammar. It starts with the first codon and the axiom of the grammar, and finishes when the genotype is exhausted or when there are no more non-terminal symbols in the current derived word. This last condition means that the mapping has derived a word belonging to the language of the grammar, i. e. a syntactically correct program.

To process each codon, the next non terminal symbol is selected from the current string (usually the leftmost one), the (n+1) rules applicable to the non terminal are enumerated (from 0 to n), and the current codon is mapped into one of them (usually by computing *codon mod n*).

1.4 Practical considerations

In the first step, this paper solves a symbolic regression problem by means of GE. The algorithm has been implemented in Java, we have tried to keep as close as possible to the original description in reference[5].

The symbolic regression problem tries to find a symbolic expression fitting a given function on a set of control points.

In Genetic Programming [6], most of the problems can be reduced to the symbolic regression problem. Therefore, this paper will be restricted to that problem. The target function chosen is $f(x) = x^4 + x^3 + x^2 + x$

The solution of this problem in [5] is done by means of the following features and parameters:

Several characteristics of our experiments are the same as in [5]:

- The context free grammar

```
<expr> ::= <expr> + <expr>
          | <expr> - <expr>
          | <expr> * <expr>
          | (<expr>)
          | <pre_op>(<expr>)
          | <var>
<pre_op> ::= sin
          | cos
          | exp
          | log
<var> ::= x
```

- The set of control points: 21 values uniformly taken from $[-1, 1]$.
- The fitness function: the sum of the absolute error over the set of the 21 control points.
- The population size: 500.
- The length of the genotypes: variable length, initially between 1 and 10.
- Codons: they belong to the $[0, 256]$ interval.
- Probability of crossover: 0.9.
- Bit mutation with probability 0.1.

We have tried to reproduce the same example, with the following differences:

- The above grammar is ambiguous, as it does not define any operator precedence. In the Java application, we have removed the ambiguity and designed the associated pushdown automata to make the evaluation of the individuals easier. For the sake of simplicity, the grammar appearing in the remainder of this paper is still the ambiguous version.
- The high cumulative success frequency described in reference[5] is not reached unless the parents are chosen with a fitness proportional strategy, and the next populations are generated by means of a generational scheme, rather than a steady state scheme.
- The size of the genotypes increase with the number of generations. It has been empirically observed that the number of unused codons also increases. The crossover operator described in reference [5] uses one single random crossover point. Thus, the number of descendants that map to the same phenotype as their parents, correspondingly increases. To solve this situation, we have restricted the crossover point choice to the used portion of each genotype, rather than to the whole genotype.
- In our work, mutation changes a single codon by a random value in the $[0, 256]$ interval. The best probability rate of mutation was found empirically as 0.5, although the performance is very similar for mutations in $\{0.5, 0.7, 0.8\}$.

2 Attribute grammar evolution (AGE)

2.1 Previous similar works

This is not the first attempt to extend genetic programming by adding the complete description of a programming language: references [7,8,9,10] describe some Prolog based approaches. These algorithms are criticized by some authors [11,12] because the logic engine makes it difficult to control some parameters of the search algorithm, and because the backtracking tends to worsen the final performance and does not ensure that the computation finishes in all the possible cases.

AGE mainly differs from the previous references in the following:

- It does not depend on any programming paradigm.
- Ross and Hussain's works represent the genotypes by means of trees.
- Man Leung Wong and Kwong Sak Leung's works are mainly interested in data mining and machine learning.

2.2 An algorithm to evaluate the attributes while building the derivation tree

Attribute grammars are exhaustively used in the design of parsers for program translators. AGE uses them to derive the phenotypes. The attributes are evaluated by means of the derivation tree. Each time that a node of the tree is expanded, the values of the attributes that can be evaluated are computed in the following way:

- Attributes inherited from the parent symbol are evaluated directly.
- If the node symbol is prefixed by other symbols to the right of where it appears, attributes inherited from the left siblings are also evaluated.
- After expanding the last child of a node, the parent synthesized attributes are evaluated.

The axiom of the grammar has only synthesized attributes. The leaves of the tree are associated to terminal symbols without any attributes of their own, but which may be used to input data into the derivation tree.

AGE uses the attributes to describe the conditions that a phenotype must comply with to be considered semantically valid. As soon as one of these constraints is violated, the derivation process is aborted.

2.3 Modifying the genotype to phenotype mapping

AGE adds the evaluation of the attributes to the previously described GE mapping. The following steps outline the algorithm applied to every codon in the genotype:

1. Choose the leftmost non-terminal symbol in the current word.
2. Select in the tree the node associated with the symbol. This is the current node.
3. Update the attributes in the derivation tree.
4. Number in zero origin the right hand sides of all the rules for this non-terminal symbol.
5. Select the right hand side of the rule whose number equals *codon mod number of right hand sides* for this non-terminal.
6. Derive the next word by replacing the non-terminal by the selected right hand side.
7. Insert a new node for each symbol to the right hand side of the rule and make all the new nodes the children of the current one.

Figure 1 graphically shows the derivation of the expression $x + \cos(x)$ from a genotype and the following attribute grammar:

```

<expr> ::= <expr>1 <op> <expr>2 { <expr>.v = <op>.f(<expr>1.v, <expr>2.v); }
        | <pre_op>(<expr>1) { <expr>.v = <pre_op>.f(<expr>1.v); }
        | <var> { <expr>.v = <var>.v; }
<pre_op> ::= sin { <pre_op>.f = cos; }

```

```

        | cos{<pre_op>.f=sin;}
        | log{<pre_op>.f=log;}
<op> ::= +{<op>.f=+;}
        | -{<op>.f=-;}
        | *{<op>.f=*;}
<var> ::= x{<var>.v=0;}
Where:

```

- Non-terminal symbols <expr> and <var> have an attribute *v*, that stands for the value of the expression. Notice that the last rule inputs 0 as the value of the variable *x*, the value of <var> attribute *v*.
- Non-terminal <pre_op> and <op> have an attribute *f* that represents the function that will be applied when computing the value of the expression.
- The association of an attribute to a symbol is represented by means of a dot as in the C language.

2.4 GE vs. AGE performance

Our first experiment avoids the generation of a phenotype that may be undefined on any control point. This can only happen when the expression contains *log* (subexpression) and the value of the subexpression is less or equal to 0.

The goal of the first experiment is estimating the possible loss of performance due to the steps added to the mapping algorithm. We are not really improving GE, because no new semantics is actually added to the algorithm. There are, however, some differences between AGE and GE in this case: GE generates semantically invalid phenotypes that will probably be punished with the worst fitness value, but AGE prevents the generation of such expressions as soon as possible.

This experiment uses the same parameters as those described for GE, except for the following:

- The attribute grammar used is the following:

```

<expr> ::= <expr>1 + <expr>2 {<expr>.vi=<expr>1.vi+<expr>2.vi ∀i∈[0,20]}
        | <expr>1 - <expr>2 {<expr>.vi=<expr>1.vi-<expr>2.vi ∀i∈[0,20]}
        | <expr>1 * <expr>2 {<expr>.vi=<expr>1.vi*<expr>2.vi ∀i∈[0,20]}
        | (<expr>1) {<expr>.vi=<expr>1.vi ∀i∈[0,20]}
        | <pre_op>(<expr>1) {<expr>.vi=<pre_op>.f(<expr>1.vi) ∀i∈[0,20]}
        | <var> {<expr>.vi=<var>.vi ∀i∈[0,20]}
<pre_op> ::= sin{<pre_op>.f=sin}
        | cos{<pre_op>.f=cos}
        | exp{<pre_op>.f=exp}
        | log{<pre_op>.f=log}
<var> ::= x{<var>.v0=-1
        <var>.v1=-0.9

```

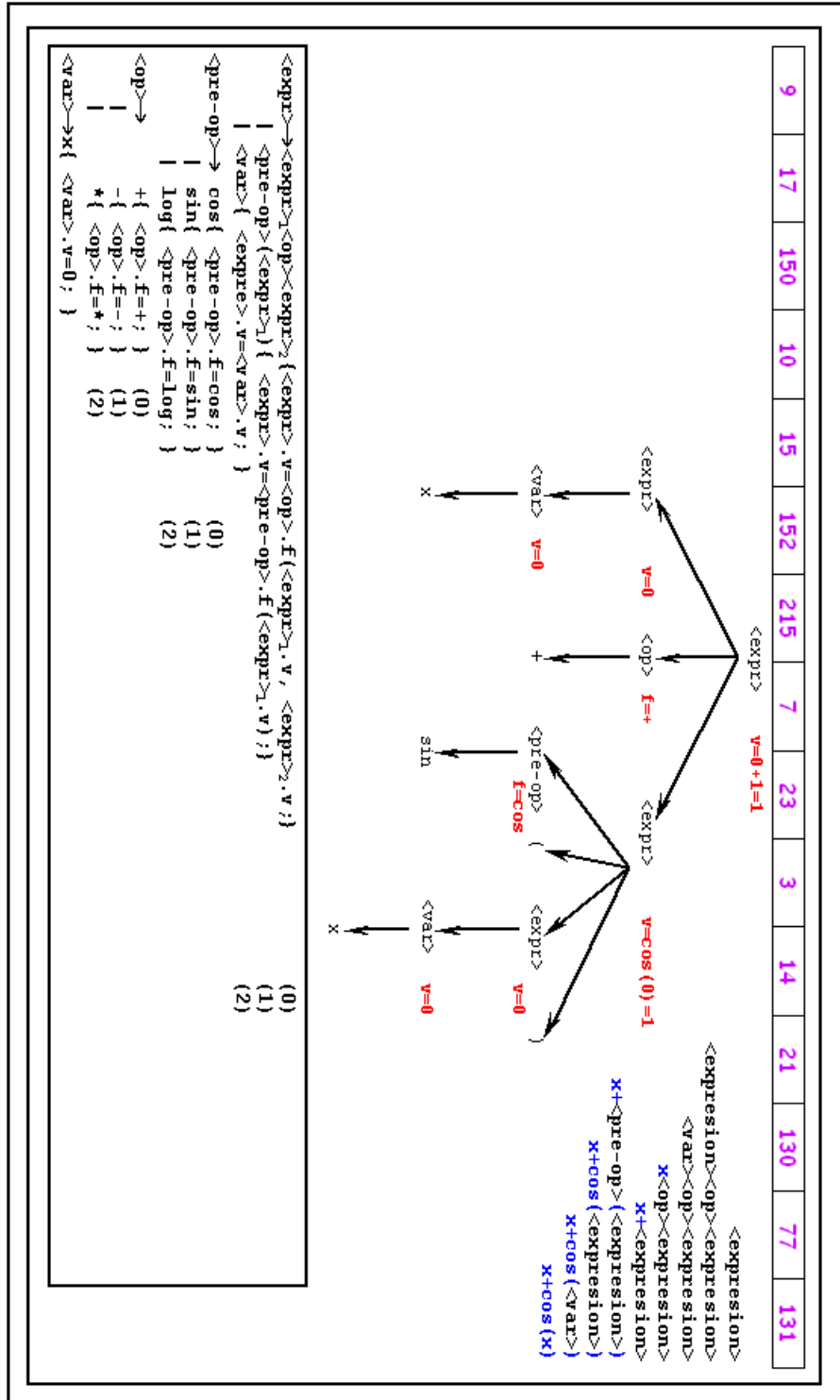


Fig. 1. AGE genotype to phenotype mapping of $x + \cos(x)$. The attribute grammar computes its value in $x = 0$

```

<var>.v1=-0.8
...
<var>.v20=1}

```

This grammar is very similar to the one used in the genotype to phenotype mapping example. The main difference is the existence of 21 attributes ($v_i \forall i \in [0, 20]$) to record the value of the expression on each control point.

- The probability of mutation has been empirically optimized to 0.7 as shown in figure 2.

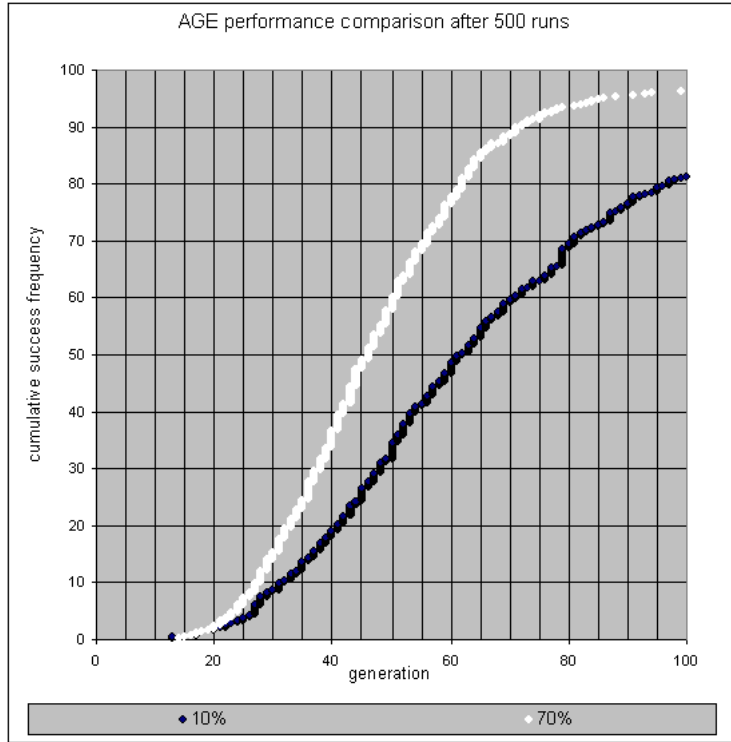


Fig. 2. AGE performance comparison after 500 runs. The worst performance corresponds to a probability of mutation equals 0.1 and the best to 0.7.

Figure 3 shows the performance comparison between GE and AGE for the best probability of mutation after 500 runs of the algorithm. As expected, there is no remarkable improvement in performance. However, there neither is any remarkable loss. So, we can deduce that AGE is as good as GE to solve this problem. As many of the problems solved by GP are reducible to the symbolic regression domain, we are optimistic about the generalization of these results to other kinds of problems.

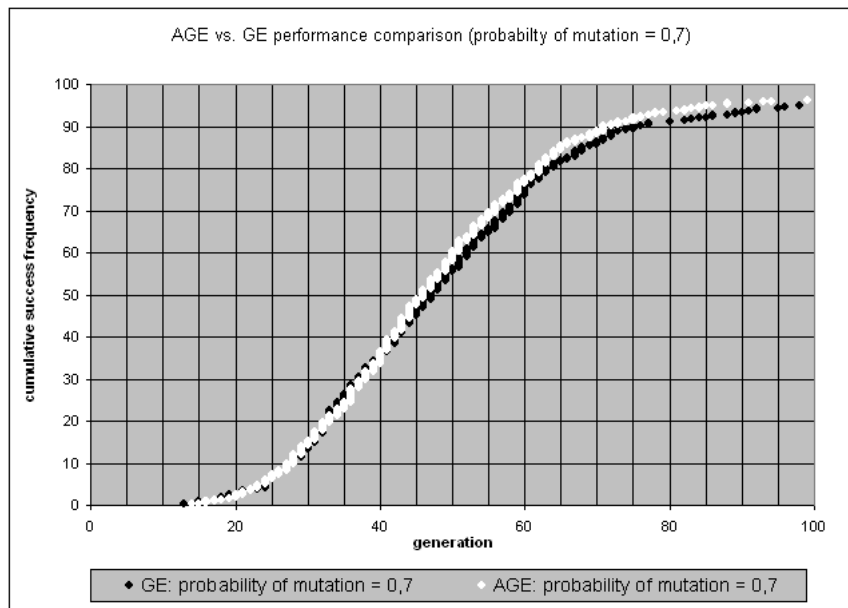


Fig. 3. GE vs. AGE performance comparison after 500 runs with the best probability of mutation (0.7). As with the rest of probabilities of mutation, there is no remarkable performance change.

The goal of our second experiment is to check if, after adding some semantics to the grammar, the algorithm improves its performance. We shall add constrains to consider invalid any phenotype that does not exactly fit the target function in any of the three control points: -1, 0 or 1.

For this experiment, the probability of mutation is set to 0.7, one of the better values both in GE and in AGE.

Figure 4 shows a significant improvement in performance after 500 runs of the algorithm: the cumulative frequency of success grows faster and reaches higher values.

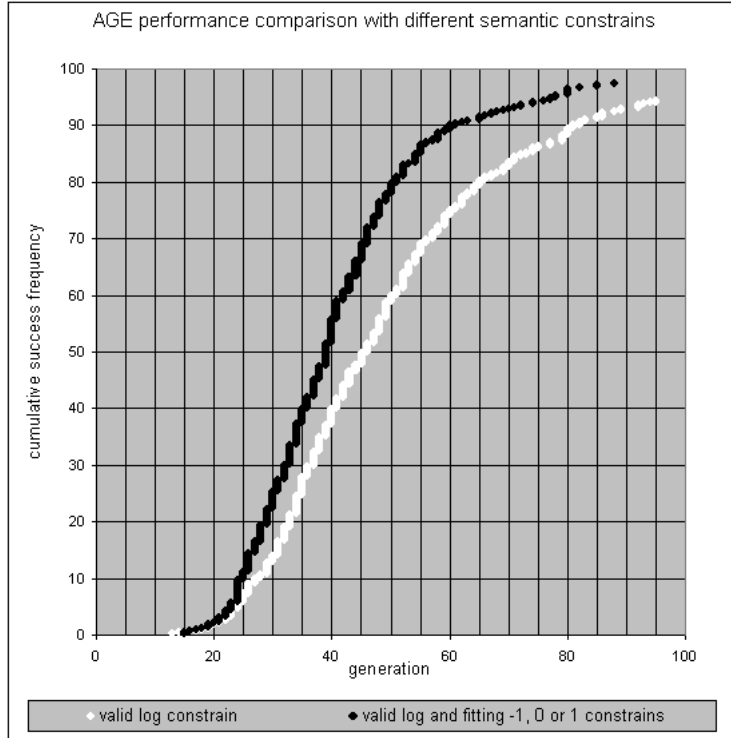


Fig. 4. AGE performance comparison with different semantic constrains. The improvement in the performance after adding a rather loose constrain is significant.

It is worth noticing that, even although the semantic constrains are rather loose, the improvement is significant. On the other hand, other tests have shown that increasing too much the semantics causes bad performance. A possible reason is that the expressive power of attribute grammars allows a full description of the solution to the problem. But, in this case, the only valid phenotypes is the solutions ,and the search is not directed by the genetic engine: it really becomes a random search.

This is a very important topic for further research: determining how many semantics must be added to get the optimal performance.

3 Conclusions and future research

This work describes AGE (attribute grammar evolution), a new automatic evolutionary programming algorithm that extends GE (grammar evolution) by adding semantic constraints which make it possible the removal from the population of, syntactically and semantically invalid phenotypes.

The differences with other approaches to genetic programming that use syntactic and semantic descriptions are discussed in the paper.

It is shown that AGE is as good as GE for a standard problem in GP. It is also proved that adding rather loose semantic constraints improves significantly the performance of AGE.

In the future, we plan to extend GE with different formal descriptions of the programming language and to apply this approach to automatically solve problems in other domains. We also plan to study in depth the determination of the adequate semantics needed to get the optimal performance.

References

1. A. N. Chomsky, Formal properties of grammars, in Handbook of Math. Psych., vol. 2, John Wiley and Sons, New York. 1963, pp. 323-418
2. D. E. Knuth, Semantics of Context-Free Languages, in Mathematical Systems Theory, vol. 2, n° 2, 1968, pp. 127-145
3. A.V. Aho, R. Sethi, J.D.Ullman (1986) Compilers: Principles, Techniques and Tools. Addison-Wesley
4. D.Grune, et al. (2000) Modern Compiler Design. John Wiley & sons Ltd.
5. M. O'Neill & R. Conor, Grammatical Evolution, evolutionary automatic programming in an arbitrary language, Kluwer Academic Publishers, 2003.
6. W.Banzaf, P.Nordic, R.E.Keller, F.D.Francone. Genetic Programming. An introduction Morgan and Kaufmann Publishers, Inc.1998
7. B.J. Ross, Logic-based Genetic Programming with Definite Clause Translation Grammars 1999
8. M.L.Wong and K.S. Leung. Evolutionary Program Induction Directed by Logic Grammars. Evolutionary Computation, 5 (2): 143-180, 1997
9. T.S.Hussain and R.A.Browse. Attribute Grammars for Genetic Representations of Neural Networks and Syntactic Constraints of Genetic Programming. In AIVIGI'98: Workshop on Evolutionary Computatino, 1998
10. T.S.Hussain. (2003) Attribute Grammar Encoding of the Structure and Behaviour of Artificial Neural Networks. Ph.D.thesis. Queens University. Kingston, Ontario, Canada
11. J. N. Shutt, Recursive Adaptable Grammars. A thesis submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, August 10, 1993 (enmended December 16, 2003)
12. H. Christiansen, A Survey of Adaptable Grammars, in ACM SIGPLAN Notices, vol. 25, n°11. November 1990, pp. 3544.